Synchronization and Deadlocks
(or The Dangers of Threading)
CS449 Fall 2016
Data Race

• **Data Race**: a situation where two threads ‘race’ to access a memory location where at least one access is a write
  – Writes to $A[tail]$ and writes to $tail$ race with each other
    (No ordering specified; outcome dependent on execution speed)
  – Writes in Thread 1 ‘disturb’ Thread 0 when it does not want to be
    (When the queue in Thread 0 is still in an inconsistent state)
  – Two reads are okay (reads do not create inconsistent states)
Critical Sections

- **Critical Section**: a section of code where a thread does not want to be disturbed, while performing a critical set of actions
  - Doesn’t want another thread to modify state that is being accessed
  - Doesn’t want another thread to access state in middle of modification
  - In other words, doesn’t want a data race
Synchronization

• Critical sections can always be disturbed
  – Two threads can be running in parallel on two CPUs
  – Even on a single CPU, context switches can happen

• Need a way to make critical sections “atomic”
  – Atom: smallest unit that cannot be divided further using chemical means
  – Atomic: property of code where the code is always executed as a unit with no interleaving

• Need help from the Operating System (or the user thread scheduler)
Mutex

- MUTual EXclusion: a lock that only one thread can acquire
- To protect a critical section:
  1. Acquire lock at the beginning
     - All other threads attempting to enter are blocked
  2. Release lock at the end
     - Another thread can now enter after acquiring lock
Critical Sections

Shared Data:

<table>
<thead>
<tr>
<th>tail</th>
<th>A[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$6$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

mutex

Enqueue():

Thread 0

lock(&mutex);
A[tail] = 20;
tail++;
unlock(&mutex);

Thread 1

lock(&mutex);
A[tail] = 9;
tail++;
unlock(&mutex);

Blocked!
#include <stdio.h>
#include <pthread.h>

int tail = 0;
int A[20];

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void enqueue(int value)
{
    pthread_mutex_lock(&mutex);
    A[tail] = value;
    tail++;
    pthread_mutex_unlock(&mutex);
}
Deadlocks

• **Deadlock**: situation where two (or more) threads cannot make progress because they are waiting for each other to release a resource (e.g. a lock)

• Caused when:
  1. Mutual exclusion on resource
  2. Hold resource and wait
  3. No preemption (reclamation) of resource
  4. *Circular wait*
Deadlock Example: Dining Philosophers

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- What happens if each philosopher grabs the fork on the right?
  1. Mutual exclusion on resource
  2. Hold resource and wait
  3. No preemption of resource
  4. Circular wait
Deadlock Example

Shared Data:

<table>
<thead>
<tr>
<th>a</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex_a</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex_b</td>
<td></td>
</tr>
</tbody>
</table>

```
Thread 0
lock(&mutex_a);
lock(&mutex_b);
a++; b++;    
unlock(&mutex_b);
unlock(&mutex_a);
```

```
Thread 1
lock(&mutex_b);
lock(&mutex_a);
b++; a++;    
unlock(&mutex_a);
unlock(&mutex_b);
```

Blocked!
Blocked!
Solving Deadlocks

- Deadlocks are caused when:
  1. Mutual exclusion on resource
  2. Hold resource and wait
  3. No preemption of resource
  4. Circular wait

- When that resource is a lock, locks by definition are mutually exclusive and allow no preemption

- So either…
  - Do not hold a lock and wait for another lock
  - Remove circular wait situations
Deadlock Solution 1: Remove Hold and Wait

Shared Data:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

mutex_a  mutex_b

```c
lock(&mutex_a);
a++;
unlock(&mutex_a);
lock(&mutex_b);
b++;
unlock(&mutex_b);
```

Thread 0

```c
lock(&mutex_b);
b++;
unlock(&mutex_b);
lock(&mutex_a);
a++;
unlock(&mutex_a);
```

Thread 1
Deadlock Solution 2: Remove Circular Wait

Shared Data:

\[ \begin{array}{c|c}
\text{a} & 0 \\
\text{b} & 0 \\
\end{array} \]

- Rule: Acquire locks in the same order

Thread 0:

\[
\begin{align*}
\text{lock}(&\text{mutex}_a); \\
\text{lock}(&\text{mutex}_b); \\
\text{a}++; \text{b}++; \\
\text{unlock}(&\text{mutex}_b); \\
\text{unlock}(&\text{mutex}_a);
\end{align*}
\]

Thread 1:

\[
\begin{align*}
\text{lock}(&\text{mutex}_a); \\
\text{lock}(&\text{mutex}_b); \\
\text{b}++; \text{a}++; \\
\text{unlock}(&\text{mutex}_b); \\
\text{unlock}(&\text{mutex}_a);
\end{align*}
\]
Is That Enough?

• Synchronization APIs seen so far
  – pthread_join: Waits for another thread to finish and (optionally) produce a value
  – pthread_mutex_lock / _unlock: Waits for another thread to finish a critical section
  – Essentially waits for a certain condition to happen
• A more general API for waiting for a condition
  – pthread_cond_wait: Waits for a condition
  – pthread_cond_signal: Signals a condition
Condition Variables

• A condition under which a thread executes or is blocked

• `pthread_cond_t`
  – The type of the condition variable

• `pthread_cond_wait (condition, mutex)`
  – Blocks current thread until condition signaled

• `pthread_cond_signal (condition)`
  – Unblocks one thread waiting for condition
Producer/Consumer Problem

Shared variables
#define N 10;

int buffer[N];
int in = 0, out = 0, counter = 0;

Producer
while (1) {
  if (counter == N)
    sleep();

  buffer[in] = ...;
  in = (in+1) % N;

  counter++;

  if (counter==1)
    wakeup(consumer);
}

Consumer
while (1) {
  if (counter == 0)
    sleep();

  ... = buffer[out];
  out = (out+1) % N;

  counter--;

  if (counter == N-1)
    wakeup(producer);
}
Producer/Consumer with only Mutexes

```c
#define N 10
int buffer[N];
int in = 0, out = 0, counter = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *producer(void *junk) {
    while(1) {
        pthread_mutex_lock(&mutex);
        if( counter == N ) {
            pthread_mutex_unlock(&mutex);
            sleep(10); // How much to sleep?
            continue;
        }
        buffer[in] = ...;
        in = (in + 1) % N;
        counter++;
        // No wakeup
        pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *junk) {
    while(1) {
        pthread_mutex_lock(&mutex);
        if( counter == 0 )
            pthread_mutex_unlock(&mutex);
        sleep(10); // How much to sleep?
        continue;
    }
    ... = buffer[out];
    out = (out + 1) % N;
    counter--;
    // No wakeup
    pthread_mutex_unlock(&mutex);
}
```

- Correct code with no data races
- But no wakeup, so must guess how much to sleep: less efficient
Producer/Consumer Final Solution

```c
#define N 10
int buffer[N];
int in = 0, out = 0, counter = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t prod_cond = PTHREAD_COND_INITIALIZER;
pthread_cond_t cons_cond = PTHREAD_COND_INITIALIZER;

void *producer(void *junk) {
    while(1) {
        pthread_mutex_lock(&mutex);
        if( counter == N )
            pthread_cond_wait(&prod_cond, &mutex);
        buffer[in] = ...;
        in = (in + 1) % N;
        counter++;
        if( counter == 1 )
            pthread_cond_signal(&cons_cond);
        pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *junk) {
    while(1) {
        pthread_mutex_lock(&mutex);
        if( counter == 0 )
            pthread_cond_wait(&cons_cond, &mutex);
        ... = buffer[out];
        out = (out + 1) % N;
        counter--;
        if( counter == (N-1) )
            pthread_cond_signal(&prod_cond);
        pthread_mutex_unlock(&mutex);
    }
}
```
Mutex in pthread_cond_wait()

- Note: a mutex is passed to pthread_cond_wait
  
  ```c
  pthread_mutex_lock(&mutex);
  if( counter == N )
  
  pthread_cond_wait(&prod_cond, &mutex);
  
  – Unlocks mutex and waits for condition in a single atomic operation
  ```

- Could we not have done something like this instead?
  
  ```c
  pthread_mutex_lock(&mutex);
  if( counter == N ) {
  
  pthread_mutex_unlock(&mutex);
  
  pthread_cond_wait(&prod_cond);
  
  }
  ```

- No. What if consumer thread intervenes between the unlock and wait, updates counter, and signals producer?
  
  – Signal would be lost, and the producer would be waiting forever

- Mutex makes check of condition and wait for condition atomic
Helgrind

- Same tool we used for memory errors
- Helgrind: component of valgrind that does potential data race / deadlock detection
- Command: valgrind --tool=helgrind <program>
- Not perfect.
  - Can miss errors (sometimes)
  - Can report errors when there are none (sometimes)
- Not a replacement for sound programming
Pitfall – Data Race on Stack

```c
void *process(void *p) {
  int socket = *(int*)p;
  ...
}

int main() {
  pthread_t thread;
  int newsocket;
  ...
  while(1) {
    newsocket = accept(socket, (struct sockaddr *)&addr, &len);
    pthread_create(&thread, NULL, process, (void *)&newsocket);
  }
}
```

- newsocket stack variable shared by all threads!
  - Main thread may overwrite newsocket due to a new connection while worker thread is still using it in process()
- Data races are not exclusive to heap and globals.
Pitfall – Data Race on Stack

```c
void *process(void *p) {
    int socket = *(int*)p;
    ...
    free(p); // don’t forget to free!
}

int main() {
    pthread_t thread;
    int *arg, newsocket;
    ...
    while(1) {
        newsocket = accept(socket, (struct sockaddr *)&addr, &len);
        arg = malloc(sizeof(int));
        *arg = newsocket;
        pthread_create(&thread, NULL, process, (void *)arg);
    }
}
```

- Now each thread has its own copy of socket on the heap
Pitfall – Too Small Critical Section

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void enqueue(int value)
{
  pthread_mutex_lock(&mutex);
  A[tail] = value;
  pthread_mutex_unlock(&mutex);
  pthread_mutex_lock(&mutex);
  tail++;
  pthread_mutex_unlock(&mutex);
}
```

- Technically removes data race (valgrind will not detect it)
- Still doesn’t prevent illegal interleavings between threads
Pitfall – Too Small Critical Section

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void enqueue(int value)
{
    pthread_mutex_lock(&mutex);
    A[tail] = value;
    tail++;
    pthread_mutex_unlock(&mutex);
}
```

• Now only consistent state is exposed to other threads
Pitfall – Too Big Critical Section

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void enqueue(int value);
void* thread_func(void*)
{
    int I, val;
    pthread_mutex_lock(&mutex);
    for(i = 0; i < 1000; i++) {
        val = ... // compute val
        enqueue(val);
    }
    pthread_mutex_unlock(&mutex);
}
```

- Contains no data race
- But too large critical section kills almost all parallelism
Pitfall – Too Big Critical Section

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void enqueue(int value);
void* thread_func(void*)
{
    int I, val;
    for(i = 0; i < 1000; i++) {
        val = ... // compute val
        pthread_mutex_lock(&mutex);
        enqueue(val);
        pthread_mutex_unlock(&mutex);
    }
}
```

• Now loop in each thread can run in parallel
Pitfall – Unnecessary Synchronization

void* thread_func1(void*) {
    pthread_mutex_lock(&mutex);
    for(i = 0; i < 1000; i++) sum += A[i];
    pthread_mutex_unlock(&mutex);
}

void* thread_func1(void*) {
    pthread_mutex_lock(&mutex);
    for(i = 0; i < 1000; i++) product *= A[i];
    pthread_mutex_unlock(&mutex);
}

• No data race so locking not needed
  – A[i] is only read in both threads
  – Neither thread disturbs each other
int done = 0, value = 0;
void* producer_thread(void*) {
    value = …;  // produce some value
    done = 1;  // signal that computation is done
}
void* consumer_thread(void*) {
    while(done != 1);  // wait until done
    ... = value;  // consume some value
}

• Looks correct as is but...
• Compiler optimization or high-performance CPU may flip the writes to ‘done’ and ‘value’
Pitfall – Jerry-Rigged Synchronization

```c
int done = 0, value = 0;
void* producer_thread(void*) {
    done = 1;  // signal that computation is done
    value = ...;  // produce some value
}
void* consumer_thread(void*) {
    while(done != 1);  // wait until done
    ... = value;  // consume some value
}
```

- Often *mistakenly* done by programmers to avoid the ‘overhead’ of POSIX calls
- Always use POSIX synchronization unless you know what you are doing (not many people in the world do)